Jotai: a Methodology for the Generation of Executable C Benchmarks

Cecília Conde Kind

UFMG, Brazil

Michael Canesche

UFMG, Brazil

Fernando Magno Quintão Pereira

UFMG, Brazil

Abstract

This paper presents a methodology for automatically generating well-defined executable benchmarks in C. The generation process is fully automatic: C files are extracted from open-source repositories and split into compilation units. A type reconstructor infers all the types and declarations required to ensure that functions compile. The generation of inputs is guided by constraints specified via a domain-specific language. This DSL refines the types of functions, for instance, creating relations between integer arguments and the length of buffers. Off-the-shelf tools such as AddressSanitizer and Kcc filter out programs with undefined behavior. To demonstrate applicability, this paper analyzes the dynamic behavior of different collections of benchmarks, some with up to 30 thousand samples, to support several observations: (i) the speedup of optimizations does not follow a normal distribution—a property assumed by statistical tests such as the T-test and the Z-test; (ii) there is strong correlation between number of instructions fetched and running time in x86 and in ARM processors; hence, the former—a non-varying quantity—can be used as a proxy for the

Email addresses: cissa.kind@dcc.ufmg.br (Cecília Conde Kind), michaelcanesche@dcc.ufmg.br (Michael Canesche), fernando@dcc.ufmg.br (Fernando Magno Quintão Pereira)

latter—a varying quantity—in the autotuning of compilation tasks. The apparatus to generate benchmarks is publicly available. A collection of 18 thousand programs thus produced is also available as a CompilerGym's dataset.

1. Introduction

Predictive compilation is a family of techniques whose goal is to let optimizing compilers treat programs differently. The predictive compiler is trained on a large corpus of programs, determining, for each one of them, the sequence of analyses and optimizations that suits that code better. Once given an unknown program, the compiler uses the knowledge acquired during training to determine the best way to treat this new code. Predictive compilation methodologies have been known for many years [1, 2, 3, 4]. Nevertheless, the growing popularity of machine learning techniques has attracted new attention to this field, and much progress in the design and implementation of predictive compilers has been attained in the last five years [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18].

The Need for Benchmarks. Training a predictive compiler requires benchmarks, as emphasized in Wang and O'Boyle's [19] survey. Thus, there has been much recent research effort along the automatic synthesis of large collections of benchmarks. In 2016, Mou et al. [12] released Poj: 104 classes of programming problems, each with 500 solutions. Later, in 2017, Cummins et al. [20] produced ClGen, a tool that synthesizes OpenCL benchmarks. In 2021, Faustino et al. [10] released AnghaBench, a suite with more than one million compilable programs. A few months later, Puri et al. [21] released CodeNet. Like Poj, the CodeNet suite consists of solutions to programming problems; however, this collection is two orders of magnitude larger. More recently, Grossman et al. [22] released a dataset with 182 Billion tokens extracted from programs in the LLVM intermediate representation. To this day, Grossman's ComPile is the largest public repository of compilable programs.

The Challenge: Sound Executable Code. Most of the research to create benchmarks orbits around C, or around similar languages, such as C++ or OpenCL. Programs written in these languages might present *undefined behaviors* [23]. Undefined behavior refers to program operations that the language specification does not define, meaning the compiler is free to handle them in any way. In other words, in face of undefined behavior, the compiler may crash, produce unexpected results, or even seem to work correctly. Languages like C, C++, and OpenCL allow undefined behavior by design: it is predicted in their standards. The design of these languages prioritizes low-level control and performance [24]. By leaving certain behaviors undefined, these languages allow compilers to generate highly optimized machine code without inserting safety checks. However, this decision also places greater responsibility on the programmer to avoid problematic constructs.

Consequently, large collections of benchmarks [10, 12, 22] reconstructed from open-source repositories are formed by programs that compile but do not run. To give the reader some perspective on the problem, notice that CompilerGym [9], a framework for the autotuning of compilation tasks, provided 1,177,462 benchmarks spread across 12 datasets in October 2023. However, before Jotai programs were incorporated into it, very few programs would yield executable binaries with ready-to-use inputs: 23 programs from CBench [25] and 40 programs from MiBench [26].

Benchmarks generated to emulate human-made code have been released recently [27, 20, 28, 15, 16, 29]. Two of these collections contained C code [29, 27], but only Berezov's ColaGen [27] could be executed automatically. These are simple kernels — nests of loops that process arrays. Although simple, every program that we tried to run showed some form of undefined behavior, once compiled with Kcc [23][1]. We also compiled Armengol's ExeBench [29] with Kcc. In this case, compilation is not automatic: in our setting (Ubuntu 22.10 with clang 15.0) we had to manually fill up missing libraries. In spite of that, all

[1]These issues were reported to Berezov et al. on August 11^{th}, 2022.

the programs that we ran contained undefined behavior. During our experience with these benchmark generators, we also had to deal with another limitation: they do not provide a way to steer the generation of program inputs. Inputs are hard-coded in the synthesizer: essentially, they consist of large buffers filled with random values. It is not possible, for instance, to establish relations between function arguments.

The Contributions of this Work. The goal of this paper is to propose a methodology to generate executable C benchmarks. This methodology is built around JotaiLang, a domain-specific language that we have designed to generate inputs for programs. The process of generating benchmarks relies on a number of techniques and tools:

Techniques: JotaiLang lets developers impose constraints on the inputs that are randomly tried on each benchmark. Constraints are derived from the signature of the target function. This combination of types and constraints prunes the space of possible inputs; hence, focusing input generation on values that are more likely to result in well-defined executions.

Tools: (i) a web crawler that retrieves C functions from GitHub; (ii) an off-the-shelf type inference engine for C, based on PsycheC [30, 31], that ensures that those functions compile; (iii) a code generator that produces drivers to run each benchmark; (iv) a Valgrind plugin based on CfgGrind [32], that measures coverage of these inputs; and (v) Kcc [23] plus AddressSanitizer [33] to detect undefined behaviors.

Throughout the process of generating benchmarks and interacting with people who use them [34, 35, 36, 37, 38, 39], we have compiled a list of requirements that these programs must meet. These requirements are rather informal, but we list them below because they will direct the design decisions that shall be discussed in the rest of this paper:

Compile-and-run: Each benchmark comes in a separate file as an independent compilation unit, with all the drivers necessary to run it.

Sound: In spite of C featuring undefined behavior, each benchmark abides by the semantics that Hathhorn et al. [23] have defined for the C programming language, as they are filtered using Hathhorn *et al.*'s tool, KCC.

Deterministic: The library that generates inputs is hard-coded in each benchmark, and uses a deterministic number generator.

Profilable: Benchmarks can contain multiple input sets. Some can be used for training and others for testing.

Visible: JOTAI benchmarks do not invoke library functions. Hence, every instruction is *visible* [40]. Thus, a sanitizer like KCC can observe the execution of every instruction when looking for undefined behavior.

Observable: Every function that makes up JOTAI returns a value. This output can be used as a way to find bugs in compilers and interpreters; for instance, via differential testing, as CSMITH does [41].

Clean: Every memory allocated by that function's driver is deallocated before termination.

Summary of Results. We have made a collection of 36,223 executable programs mined from GPL repositories publicly available. Since October 2022, 18,761 of these functions are available as a COMPILERGYM dataset [9]. Nevertheless, JOTAI benchmarks are extracted from open-source repositories and there is no limit to how many programs can be created via the methodology that Section 3 introduces. Section 4 describes some uses of the JOTAI collection. Section 4.1 shows that it is fair to expect the construction of a valid benchmark (no undefined behavior detected by KCC) for each 34-35 functions that we find in C files from open-source repositories. Section 4.2 analyses the speedup of optimizations on different subsets of the JOTAI collection and on SPEC CPU2017. Such speedups do not follow a normal distribution, which is assumed in several statistical tests. Section 4.3 observes a strong correlation between the running time of programs and the number of instructions fetched on Intel i7 and on ARM

A15 processors. Correlation holds when programs are compiled with different optimization levels. Finally, Section 4.4 discusses the structural properties of the programs. On average, functions tend to comprise four to six basic blocks, and about half the functions have their control-flow graphs fully covered by inputs that we generate.

This Paper in Perspective. This work is an extended version of a paper originally published in the Brazilian Symposium on Programming Languages (SBLP 2023) [42]. That early work of ours—available in Portuguese only—described the preliminary implementation of the ideas present in this paper. Since that early publication, JOTAI benchmarks have seen usage in other works [43, 37]. The process to generate executable code has received contributions from outside our research group. Such contributions resulted in a new constraint-based language to steer the production of compilable programs, which has made it possible to carry out larger and more comprehensive experiments using the JOTAI collection. Currently, JOTAI is a consolidated collection of benchmarks, sufficiently useful to be available in a framework such as COMPILERGYM, for instance.

2. The Anatomy of a Benchmark

JOTAI benchmarks are produced out of programs mined from open-source repositories. The generation of benchmarks works by: (i) extracting C files via a web crawler; (ii) splitting functions in each C file into single files; (iii) inferring types for each benchmark file; (iv) generating a driver for each compilable benchmark file; (v) filtering out inputs that lead to undefined behavior. Figure 1 shows these different steps, and Example 1 illustrates the first steps of this process.

Example 1. *Figure 2 shows a function from* `status.c`*, a file taken from the* `sqlite` *repository. There are eight functions with a body within* `status.c`*. JOTAI tries to produce a benchmark out of each one of them. The process starts with code extraction: function* `countLookasideSlots` *is placed into a separate*

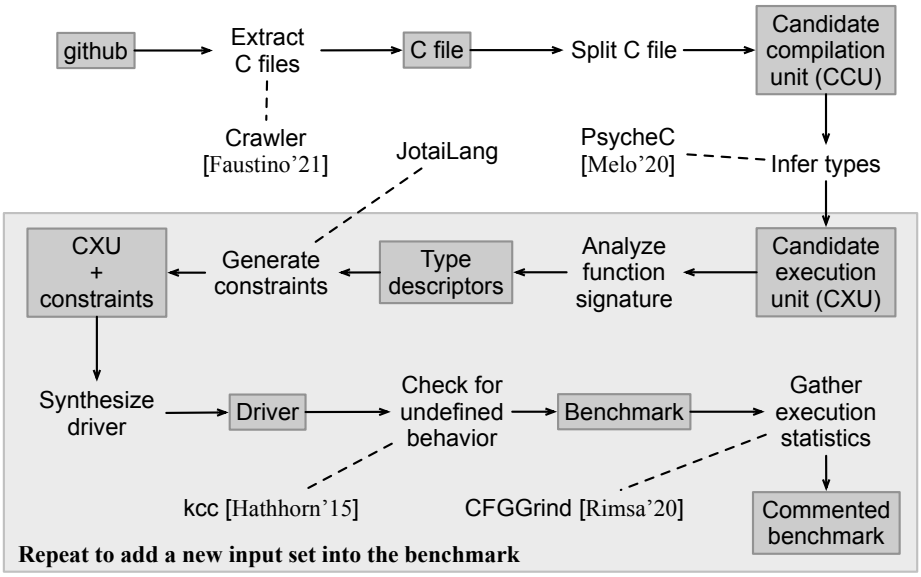

Figure 1: The benchmark generation process. The contributions of this work are separated into the gray box. This process is entirely automatic and requires starting one single script.

C file: the candidate compilation unit *(CCU). Types and missing declarations are then inferred for this CCU via a tool called* PsycheC *[30, 31]. Figure 2 (b) shows the types inferred for function* `countLookasideSlots`*. If* PsycheC *terminates successfully, then the reconstructed program is guaranteed to compile. However,* PsycheC *might fail if it receives a program that does not contain valid C syntax. Source files with unprocessed macros are the main reason why invalid syntax might occur. In this case, the candidate compilation unit is discarded.*

Visible Instructions. Example 1 illustrates one of the principles of Jotai: benchmarks yield only *visible instructions*. Following the terminology of Alvares *et al.* [40], given a program P with source code S, and a compiler C, the visible instructions of P are the instructions that C produces for statements that appear in S. Every other instruction required for the execution of P is an *invisible* instruction. Invisible instructions come from dynamically linked libraries and routines added by the compiler, such as initialization (pre-`main` code) and finalization (post-`main` code). To meet this visibility requirement,

```
175 ...
176 static u32 countLookasideSlots
                    (LookasideSlot *p) {
177   u32 cnt = 0;
178   while( p ){
179     p = p->pNext;
180     cnt++;
181   }
182   return cnt;
183 }
184 ...
```

(a)

```
/* Forward declarations */
typedef struct TYPE_3__  TYPE_1__;

/* Type definitions */
typedef  long u32 ;
struct TYPE_3__ {
  struct TYPE_3__* pNext;
};
typedef  TYPE_1__ LookasideSlot;
```

(b)

Figure 2: (a) Code snippet taken from file `status.c` from the `sqlite` repository. (b) Types that PsycheC infers to ensure compilation of function `countLookasideSlots`.

Jotai benchmarks do not invoke functions without bodies. This restriction is implemented when candidate compilation units are filtered, at the parsing level of this process. It serves two purposes. First, it eases the task of discovering undefined behavior during the execution of the program, as tools like Kcc or FramaC only have access to the visible part of a program. Second, it prevents the benchmark from invoking malicious code.

Drivers. A candidate compilation unit that compiles becomes a *candidate executable unit* (CXU). If Jotai succeeds in producing an input for a CXU that does not incur undefined behavior, then this program becomes what we call a 1-input *driver*. These benchmarks can be augmented gradually. If Jotai succeeds in generating a new input for an n-input driver, then this program becomes an $(n + 1)$-input driver. Input generation is steered by constraints, which, in turn, Jotai derives from the type signature of the target function. Constraint generation is the subject of Sections 3.1 and 3.2. For now, it suffices to know that each set of constraints that yields a well-defined execution contributes one input to the driver. Example 2 clarifies this terminology.

Example 2. *Figure 3 shows a 2-input driver produced to run the function in Example 1. This driver contains a switch with two cases: each case feeds the function* `countLookaside Slots` *with different inputs. Everything in Figure 3 is synthesized automatically from the constraints in Section 3.2.1.*

```
01  …
02  int main(int argc, char *argv[]) {
03    if (argc != 2) {
04      usage();
05      return 1;
06    }
07    int opt = atoi(argv[1]);
08    switch(opt) {
09      case 0: { // int-bounds
10        struct TYPE_3__ * aux_p[1];                          Generate inputs
11        struct TYPE_3__ * p = _allocate_p(1, aux_p);
12        long benchRet = countLookasideSlots(p);              Execute
13        printf("%ld\n", benchRet);                           Observe
14        _delete_p(aux_p, 1);                                 Clean up
15        break;
16      }
17      case 1: { // linked
18        struct TYPE_3__ * aux_p[10000];                      Generate
19        struct TYPE_3__ * p = _allocate_p(10000, aux_p);     inputs
20        long benchRet = countLookasideSlots(p);              Execute
21        printf("%ld\n", benchRet);                           Observe
22        _delete_p(aux_p, 10000);                             Clean up
23        break;
24      }
25      default:
26        usage();
27        return 2;
28    }
29    return 0;
30  }
31  …
```

Figure 3: The driver produced by JOTAI to run function `countLookasideSlots`, seen in Figure 2.

Compile-and-Run. Example 2 illustrates some principles enumerated in Section 1. First, concerning compilation, assuming that this two-input driver is in a file called `driver.c`, we can compile it with simply "`clang/gcc driver-c`". Concerning execution, we can run the executable file with either the command "`./a.out 0`", or the command "`./a.out 1`". If invoked without arguments, then the driver outputs a usage guide with a brief explanation about each input set.

Profile. The driver seen in Example 2 features two sets of inputs. Often, JOTAI benchmarks provide more than that. Multiple inputs let developers use the JOTAI benchmarks in a profile-guided setting, where some inputs are used as training, and others as testing.

Observe. The driver is also observable, meaning that it prints the result of the function. In this simple example, the output of interest is a simple scalar value; however, JOTAI benchmarks can print the contents of aggregate types, such as instances of `struct` or `union` types. Notice that inspection is shallow: recursive types like linked lists or trees are not traversed. JOTAI can also be configured to add timing routines to print the execution time of the target function; however, this code is not portable across operating systems; hence, it is disabled by default.

Clean up. Every `case` of a driver ends with calls to a routine that frees allocated memory. This cleanup is only necessary when benchmarking functions whose signature contains arguments of pointer type. Notice that JOTAI is able to generate recursive data structures, as in the second input set of Figure 3. Cleaning code will free every node that constitutes a recursive data structure. As a consequence, every JOTAI benchmark runs until normal termination (exit code 0) when compiled with an address sanitization; for instance, via `gcc -fsanitize=address`.

Deterministic behavior. JOTAI benchmarks are deterministic, meaning that the execution of an n-driver with a certain argument $i, 1 \leq i \leq n$, will always lead to the execution of the same sequence of instructions. To ensure determinism, benchmarks are sequential programs: no multi-threading is allowed. Furthermore, the routines that generate inputs for the benchmark are hard-coded into the driver. These routines include code to produce scalars of every primitive type in the ANSI C language specification, and code to generate recursive data structures like lists and trees. Function `countLookasideSlots` in Example 1 contains one argument of a recursive type, as seen in Figure 2 (b). The second case in Figure 3 will generate a linked list with 10,000 nodes with this structure.

3. A Methodology to Generate Benchmark Inputs

Figure 1 shows the steps to generate benchmarks. Part of this methodology (outside the gray box), has been already used in previous work [10], and we

omit it from this presentation. The rest of it is the subject of this section.

3.1. Extraction of Type Descriptors

The constraints that JOTAI produces to guide the generation of inputs—to be explained in Section 3.2—rely on *type descriptors*. Type descriptors model the structure of the types of the arguments of the CCU function. Descriptors are specified by the grammar in Figure 4. Following C's static semantic, type descriptors determine a nominal type system, e.g., two types are the same if they share the same names.

typeDesc	::=	(typeStruct | typeFun)*
typeStruct	::=	`struct` <*name*> typeBindings
typeFun	::=	`function` <*name*> typeName typeBindings
typeBindings	::=	<*name*> typeName typeBindings | ε
typeName	::=	(`unsigned`)? typeScalar | `*` typeName | `struct` <*name*>
typeScalar	::=	`char` | `int` | `short` | `long` | `float` | `double`

Figure 4: The grammar to specify type descriptors.

Example 3. *Figure 5 (b) shows the type descriptors that* JOTAI *extracts for the function* `sum`*, in Figure 5 (a). Notice that the definition of* `struct S` *was inferred by* PSYCHEC *in the previous stage, as explained in Example 1. The descriptors expose the structure of aggregate type* `struct S`*, and list the types used in the signature of function* `sum`*.*

Type descriptors are extracted from candidate compilation units via a `clang` plug-in implemented with the "RecursiveASTVisitor". This `clang` class is used to traverse and extract information from the Abstract Syntax Tree of C programs. Once type descriptors are extracted, the process of generation of inputs no longer uses the target function. In other words, this function is analyzed only during the extraction of type descriptors. After this point, the function is treated as a black box.

(a)
```
01 struct S {
02   int data;
03   char flag;
04 };
05 typedef struct S MyStruct;
06 void sum(MyStruct s, int* p, int n) {
07   int sum = 0;
08   if (s.flag == 's') {
09     for (int i = 0; i < n; i++) {
10       sum += p[i];
11     }
12   }
13   s.data = sum;
14 }
```

(b)
```
struct S
 | data int
 | flag char

function sum void
 | s struct S
 | p int*
 | n int
```

Figure 5: Given the function `sum` in part (a), JOTAI extracts the type descriptors in part (b) of the figure.

3.2. JOTAILANG: The Constraint Generation Language

Type descriptors are used as inputs in the process of generating *constraints* for a program. Constraints can be viewed as *refined types* [44]. In other words, instead of saying that the type of a variable is an integer, we say that this type is an integer larger than zero, for instance. Figure 6 shows the grammar of JOTAILANG, the constraint generation language. Constraints define essentially two properties over variables: `value` and `length`. The former applies to any variable; the latter only to variables of pointer types. JOTAILANG also defines a few "algorithmic skeletons", which we shall explain in Section 3.2.2. Example 4 provides examples of constraint-based refinements.

```
constraint ::= comp ( ',' comp )*

comp       ::= arith (== | != | > | < | >= | <=) arith
             | skeleton

arith      ::= element (* | + | / | - | %) element
             | element

skeleton   ::= linked (<name> ',' <integer>)
             | dLinked (<name> ',' <integer>)
             | binTree (<name> ',' <integer> ',' <integer>)

element    ::= const | (value | length) '(' variable ')'

variable   ::= <name> ( '[' ( <integer> | <name> | '_' ) ']' )*
             | <name > ( '.' <name> )*

const      ::= <integer> | <floating-point> | <char> | <string>
```

Figure 6: JOTAILANG: The constraint generation language.

Example 4. *Consider n and p in the type descriptors of Figure 5 (b). The type of n is an integer, and p has a pointer type. Constraints let us:*

- *Relate n's value with a constant, e.g.,* $\texttt{value}(n) == 100$.
- *Relate two variables, e.g.,* $\texttt{length}(p) == \texttt{value}(n)$.
- *Constrain values within buffers, e.g.,* $\texttt{length}(p) == \texttt{value}(n) + 1$, *and* $\texttt{value}(p[n]) == \texttt{'\0'}$.
- *Associate lengths of arrays and values of scalars with specific constants, e.g.,* $\texttt{value}(n) == 255, \texttt{length}(p) == 65,205$.

3.2.1. Generating Constraints

JotaiLang is not designed to be written manually by developers, because this process would be tedious and error-prone, if necessary to produce inputs for a large number of different compilation units. Instead, constraints are automatically generated through Python scripts that extend Jotai's constraint generation module. To use this module, users write Python code that explores the space of valid constraints over type descriptors. The module provides high-level constructs such as quantifiers (e.g., "for all") and existentials (e.g., "exists"), which operate over scalar, pointer, and aggregate types extracted from candidate functions. The output of this process is a set of textual constraints in the JotaiLang format, which is then passed to Jotai's input generator.

Example 5. *Figure 7 illustrates four different constraint generation methods corresponding to the refinements introduced in Example 4. These methods are implemented using the Python API provided by Jotai. Each method receives a list of constraints to be augmented and a type descriptor, which describes the variables in the function signature. The terminals defined in Figure 6 are represented as Python classes and can be combined to build complex constraints.*

Each constraint generation script produces one input for a benchmark that matches its associated type descriptors. However, a single benchmark may be paired with multiple scripts, as shown in Example 5, resulting in multiple input

The value of every scalar in the list of parameters must be 100:

```
def intBounds(ctrs, type_desc):
  for name in type_desc.scalars:
    ctrs += Value(name, 100)
```

For every scalar n and pointer p, p points to a buffer of size n+1, and p[n] == '\0':

```
def zeroEnd(ctrs, type_desc):
  for n in type_desc.scalars:
    for p in type_desc.pointers:
      c0 = Value(n, Plus(n, 1))
      c1 = Length(p, Value(n)+1)
      c2 = Value(Arr(p, Value(n)), '\0')
      ctrs += [c0, c1, c2]
```

For every scalar n and every pointer p, we have that p points to a buffer of size n:

```
def eqValLen(ctrs, type_desc):
  for n in type_desc.scalars:
    for p in type_desc.pointers:
      ctrs += Length(p, Value(n))
```

For every scalar n, n == 255, and for every pointer p, p points to a buffer of length 65,025:

```
def bigArr(ctrs, type_desc):
  for name in type_desc.scalars:
    ctrs += Value(name, 255)
  for name in type_desc.pointers:
    ctrs += Length(name, 65025)
```

Figure 7: Examples of four user-defined constraint generation methods. Complementing the schema in Figure 1, for each candidate execution unit, type descriptors are extracted via a `clang` plugin and stored in textual format. This text is loaded by a Python driver into a `type_desc` data structure. The driver then calls back user-defined functions like `intBounds` or `eqValLen` to generate constraints. Each such function receives two arguments: the list `type_desc` of type descriptors, and a list `ctrs` that accumulates constraints. These functions can iterate over the `type_desc` list and instantiate `Value` and `Length` objects to create new constraints, which are appended into the `ctrs` list.

sets. Likewise, the same script can be reused across different benchmarks, provided the type descriptors are compatible. This matching process is fully automatic. As a result, most JOTAI benchmarks evaluated in Section 4 include more than one input. Each benchmark is compiled into a standalone executable that accepts a single integer argument to select an input set—for example, "`./a.out 0`" runs the first input, while "`./a.out 1`" runs the second if available.

3.2.2. Algorithmic Skeletons

It is possible to use `length` and `value` constraints to define recursive data structures such as linked lists and trees. However, we found it easier to define a small library of algorithmic skeletons to specify data structures. Currently, we define three skeletons, which Figure 6 shows: `linked`, for linked lists; `dLinked` for doubly linked lists; and `binTree` for binary trees.

Currently, users have no control on how skeletons are used: they are hard coded in the constraint generation engine. These skeletons are chosen according to the type descriptor extracted from a candidate execution unit. If the de-

scriptor contains one recursive reference, JOTAI tries to use `linked` to generate linked lists. If the descriptor contains two recursive references, then JOTAI uses `dLinked` or `binTree` to generate data structures.

3.3. Filtering Out Undefined Behavior

There exist ANSI C programs that can be compiled by any compiler that conforms to the different C Standards, but whose runtime behavior is undefined. Quoting Hathhorn et al. [23]: "*The C11 standard mentions situations that lead to undefined behavior in 203 articles*". Among such situations, 77 involve aspects of the C language itself; i.e., are produced by the use of grammatical constructions of the language. Another 24 undefined behaviors are caused by omissions by the parser or the preprocessor. And there are 101 undefined behaviors caused by misuse of functions and variables from the language's standard library [45] (Appendix J).

The benchmarks that we produce are reconstructed from programs available in open-source repositories. Thus, undefined behaviors present in these programs are likely to persist in their benchmark versions. Additionally, our type reconstructor might introduce undefined behavior into programs that were originally correct, as Example 6 shows.

Example 6. *Figure 8 shows a program whose types were reconstructed by the type inference engine in PSYCHEC. PSYCHEC reconstructs* `u64` *and* `s64` *as "*`int`*". This is the default strategy used by the inference engine when it does not find enough constraints to determine a type precisely [30, 31]. However, the former was originally declared as "*`unsigned long`*", and the latter as "*`long`*". In this case, the left shift on Line 04 in Figure 8 yields undefined behavior, because the shift count exceeds the width of the type that PSYCHEC has inferred. The program in Figure 8, when compiled with* `gcc -O0`*,* `gcc -O1` *and* `clang -O0` *outputs the same value. However, if compiled with* `clang -O1` *then it produces a different value when running on OSX 11.2.*

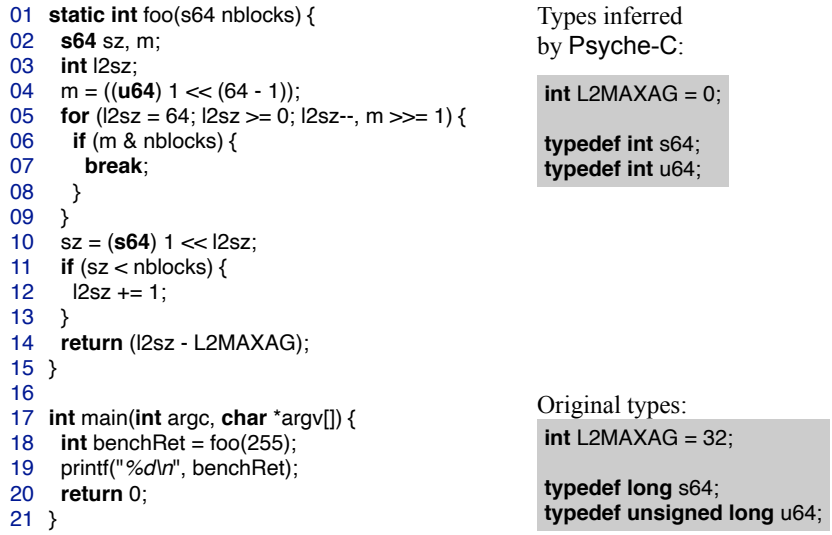

Figure 8: The left shift in Line 04 causes undefined behavior in this program.

3.3.1. AddressSanitizer and KCC

Detecting undefined behavior is not easy. Indeed, with the current technology presently available, it might be impossible. As pointed out by Memarian et al. [46]: "*The divergence among the de facto and ISO standards, the prose form, ambiguities, and complexities of the latter, and the lack of an integrated treatment of concurrency, all mean that the ISO standard is not at present providing a satisfactory definition of C as it is or should be.*" Thus, to filter out undefined behavior, we adopt a best-effort approach, based on a combination of two tools: AddressSanitizer [33] and Kcc [23]. AddressSanitizer is a fast runtime tool that detects memory-related errors such as buffer overflows, use-after-free, and memory leaks during program execution. Kcc is a formally verified C compiler that checks for undefined behavior by symbolically executing programs according to a precise model of the C standard.

We compile every candidate execution unit with `clang` and `gcc`, using, in both cases, the following flags: "`-fsanitize = address, undefined, signed -integer-overflow -fno -sanitize-recover=all`". These flags invoke different extensions from AddressSanitizer [33]. However, even the programs

that run until normal termination (exit code zero), and that output the same results with either `clang` or `gcc` can still exhibit undefined behavior, as Example 7 illustrates.

Example 7. *Consider the following program:*

```
int main() {int i = 3; i = i++; return i;}
```

This program runs until normal termination when it is compiled with the AddressSanitizer *plugins. However, once compiled with* Kcc*, it stops with the error code* `UB-EIO8`*, i.e.: "*Unsequenced side effect on scalar object with side effect of same object*" (see C11's Section 6.5:2).*

Thus, to further remove malformed benchmarks, the programs that pass through the AddressSanitizer sieve are then compiled with Hathhorn et al.'s [23] Kcc. Kcc and AddressSanitizer detect different kinds of undefined behavior because they use different approaches to do it. Kcc is a formally verified tool that models the C standard closely and checks for undefined behavior through symbolic execution and static analysis, making it effective at catching violations even without running the program. In contrast, AddressSanitizer is a fast, practical runtime tool that detects memory-related errors like buffer overflows and use-after-free, but only on paths that are actually executed. Because of these differences in design and coverage, neither tool is a superset of the other. Thus, using both leads to broader detection of undefined behaviors in C programs.

We run Kcc with a timeout of five seconds. The choice of five seconds is arbitrary; we chose it because a slower timeout could reduce Jotai's throughput, whereas a shorter timeout could yield more programs with undefined behavior. Nevertheless, using a timeout is paramount with Kcc, because it slows down the target programs by a substantial margin. Our experience with Kcc has presented us with situations where programs that do not seem to contain any undefined behavior would loop forever. Nevertheless, Kcc detects more occurrences of undefined behaviors than AddressSanitizer.

3.3.2. Freeing Memory

The `length` constraint, described in Section 3.2, causes JOTAI to generate code that allocates memory for pointer variables. Memory allocation also occurs when using the built-in skeletons described in Section 3.2.2. The `-fsanitize=address` flag, mentioned in Section 3.3.1, detects memory leaks and will terminate programs that do not properly deallocate memory.

To prevent memory leaks, JOTAI's input generator maintains a table that records all memory blocks it allocates. Before the benchmark terminates, this table is traversed and each recorded block is freed. This mechanism guarantees the proper deallocation of memory, including that used for recursive data structures generated by algorithmic skeletons.

In practical terms, this approach introduces a form of garbage collection to JOTAI benchmarks. Unlike traditional tracing garbage collectors (e.g., mark-and-sweep [47]), this approach does not reclaim memory dynamically during execution. Instead, all allocated memory is released in a well-defined cleanup phase at the end of the program, ensuring predictable and deterministic behavior. This strategy resembles region-based memory management, as used in the MLKIT compiler [48], where all allocations within a region are tracked and deallocated together. It also bears similarities to arena allocation in C/C++, in which memory is drawn from a pool and freed in bulk once the pool is no longer needed [49].

4. Evaluation

This section analyzes five research questions related to the benchmarks produced via the JOTAI methodology.

4.1. The Benchmark Generation Rate

JOTAI benchmarks are produced out of programs publicly available in open-source repositories. Thus, the number of possible "seeds" for benchmarks is virtually unbounded. However, most of the functions in these repositories will

not yield valid benchmarks. This section analyzes the rate at which benchmarks can be produced, in order to answer Question 1.

Question 1 (RQ1). *What is the ratio of candidate functions to viable programs generated by the methodology in Fig. 1?*

Benchmarks: To answer Question 1, we apply the methodology from Figure 1 onto the GPL 3.0 benchmarks publicly available in the AnghaBench repository on August 7th, 2022 (`https://github.com/brenocfg/AnghaBench`). We chose AnghaBench as the ground truth for this experiment for practical reasons: each benchmark in that collection consists of a single C function within a C file that can be compiled without any dependencies. Thus, pragmatically, it is easy to extract this function from that compilation unit in order to build a corresponding Jotai benchmark.

At the time we run this experiment, the AnghaBench repository had 1,041,333 compilable C functions taken from 148 GitHub repositories, sorted by the number of stargazers. Out of this lot, 70,309 functions are *leaf routines*; that is, functions that do not invoke other functions. We only apply the Jotai methodology to the leaf routines, to ensure the absence of invisible instructions in the benchmarks.

Hardware: Intel i7-6700T with 7.6GB of RAM.

Software: Benchmarks are compiled with `clang` 15.0 plus AddressSanitizer and with Kcc 3.4.

Methodology: We apply six constraints on each leaf routine. Three constraints are the built-in skeletons **linked** (**lk**), **dLinked** (**dl**) and **binTree** (**bt**). The former is applied onto functions containing an argument whose type has a recursive reference; **dl** and **bt** are applied onto types that contain two recursive references. The last three constraints are:

int-bounds (ib): for every scalar n: $\texttt{value}(n) = 100$.

big-arr-10x (bx): for every scalar n: $\texttt{value}(n) = 10$; and for every pointer p: $\texttt{length}(v) = 100$.

big-arr (ba): for every scalar n: $\texttt{value}(n) = 255$; and for every pointer p: $\texttt{length}(v) = 65,025$.

Discussion: Figure 9 shows the number of valid compilation units produced with AddressSanitizer and with Kcc. Out of 1,041,333 functions in Angha Bench, 70,309 were candidate compilation units, leading, in the end, to 31,328 benchmarks that could be successfully compiled with Kcc and instrumented with AddressSanitizer, running without triggering assertions. The success rate of Kcc depends on a timeout. If we compile the first 1,000 compilation units produced via **int-bounds** with Kcc, using a timeout of one second, then we obtained 937 valid programs. Increasing this timeout to 10 seconds adds one more program to this collection. Kcc failed to compile 17 programs; 46 other programs stopped with clear error messages referring to the C11 Standard.

	ib	ba	bx	lk	dl	bt	#f
Asan	40,062	39,448	39,884	192	69	97	41,995
Kcc	14,468	28,800	29,238	172	70	98	31,328

Figure 9: Number of valid execution units produced with the AddressSanitizer and the Kcc sieves. Results are cumulative: a program that passes the Kcc sieve has also passed the AddressSanitizer sieve. **#f** is the total number of benchmark files produced. Each file contains at least one and at most six different inputs.

4.2. Normality

The goal of this section is to demonstrate how Jotai can be used as a means to analyze and understand the dynamic behavior of programs. Several statistical tests (e.g.: the T-test, the Z-test, Pearson's Correlation, etc) assume that data comes from a normal distribution. The normal distribution typically emerges from the accumulation of effects produced by independent events [50]. Given that the different optimization levels of a compiler are formed by the combination of different optimizations, one could be tempted to believe that optimization speedups obey a normal distribution. This section evaluates this hypothesis.

Question 2 (RQ2). *Does the speedup observed after the application of compiler optimizations follow a normal distribution?*

Benchmarks: We evaluate Question 2 onto three different collections of programs:

SameRes: The 19,211 execution units produced via the **big-arr** constraint that return a primitive value (`int`, `uint`, `float`, etc).

DynGr: The 856 execution units produced with **big-arr** whose number of instructions executed is larger than the number of different instructions fetched. These are programs containing loops that run at least twice.

Spec: The 43 programs from SPEC CPU2017 [51]. We use **Spec** to give the reader some perspective on how Jotai programs compare with this well-established benchmark suite.

Hardware: Intel i7-6700T with 7.6GB of RAM.

Software: Benchmarks are compiled with `clang` 15.0. We count instructions using Valgrind [52]'s CfgGrind [32].

Methodology: We define *speedup* as the rate of instructions executed by the benchmark once compiled with `clang -O0` and `clang -OX`, where `X` $\in \{1, 2, 3, s, z\}$[2]. We measure speedup using the number of instructions counted by CfgGrind, instead of using running time, because this metric is stable—running time is subject to much variation. Section 4.3 provides evidence that this methodology is sound.

Discussion: Figures 10, 11 and 12 summarize the results observed in this experiment[3]. Each figure shows a density (left) and a quantile-quantile (right) plot. The former highlights means and variances in the measured speedups. The latter compares the observed distribution with a normal distribution with similar parameters. The gray area in the QQ-plot delimits the area where normal data

[2]To keep the presentation short, we plot speedups relative to `clang -O2`; however, we could observe very similar results for the other levels.

[3]To ease visualization, we crop the speedup at 8.0x in every density plot.

would be expected to exist. In all three cases, including SPEC CPU2017, the Shapiro-Wilk Normality Test returns a p-value lower than 0.0001, indicating that the observed speedups are very unlikely to come from a normal distribution.

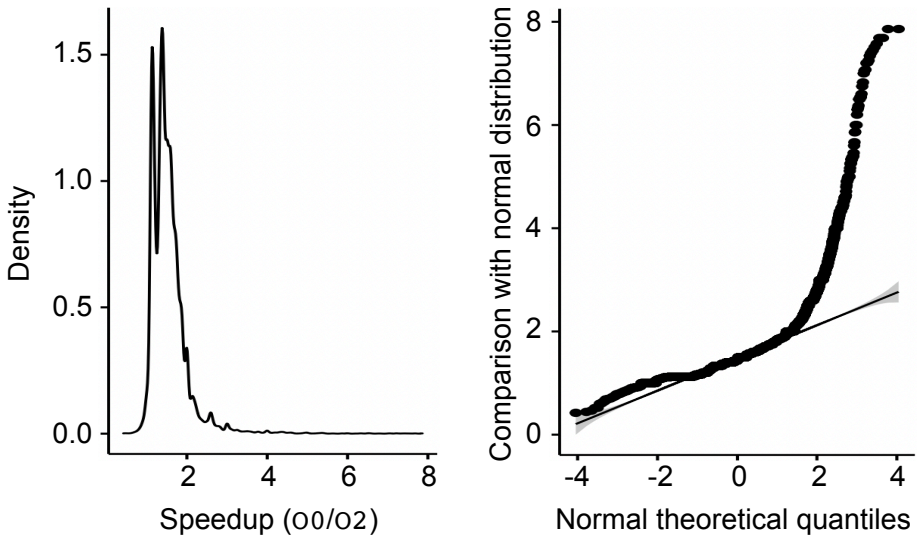

Figure 10: Density plot and QQ-plot for the 19,163 programs in the **SameRes** suite of benchmarks. The density plot is cropped at 8.0 to improve visualization. Mean = 1.54x, Median = 1.45x, Outliers (speedup greater than 8.0) = 130.

Optimizations bear stronger effects in programs containing loops, as the 90/10 Rule of Code Optimization implies [53, Ch.3]. Thus, speedups are higher on **DynGr** (Median = 2.57x) and **Spec** (Median = 2.72x) than on **SameRes** (Median = 1.45x). Nevertheless, even **SameRes** contains samples whose speedup would be impossible under a normal distribution. For instance, `clang -O2` produces a speedup higher than 500x in two programs of **SameRes**. The probability of observing this speedup under the assumption of a normal distribution is zero for all practical purposes. Indeed, it is easy to write programs where the speedup obtained after standard optimizations can be as large as one wants. As an example, both `clang` and `gcc` are able to replace loops that sum arithmetic progressions with $O(1)$ formulae.

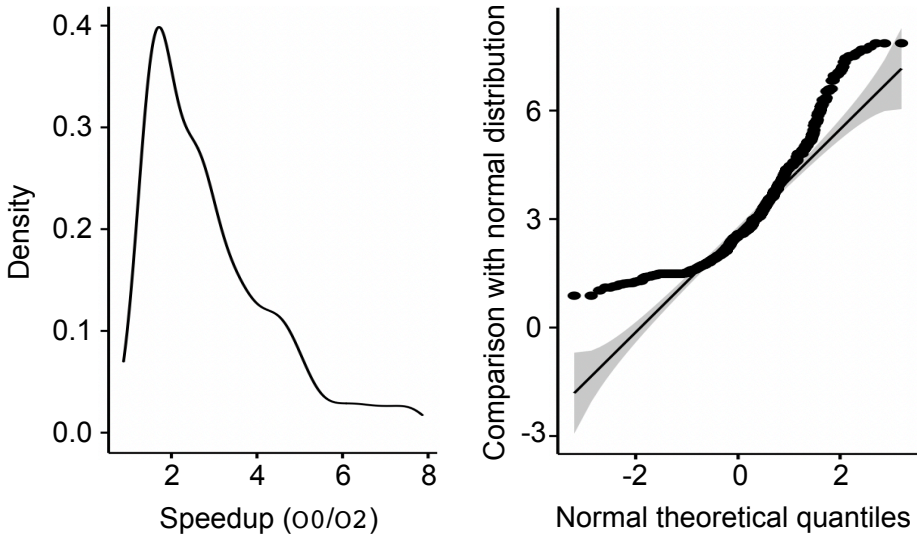

Figure 11: Density plot and QQ-plot for the 856 programs in **DynGr**. Mean = 2.91x, Median = 2.57x, Outliers = 48.

4.3. Running Time vs Instructions Fetched

Many works that predict compiler performance [54, 9] focus on code size rather than running time, as the former yields exact values, while the latter is subject to stochastic variation. These fluctuations make it challenging to use benchmarks such as JOTAI for compiler tuning, as timing measurements become imprecise. This imprecision is typically reflected in high coefficients of variation; that is, a high ratio between the standard deviation and the mean. As a result, differences in the running times of the same benchmark may be indistinguishable when analyzed with statistical tests. For instance, using the p-value as a test statistic may lead to inconclusive results. This problem is more pronounced in fast-executing benchmarks, where even small variations can hide meaningful differences. Example 8 illustrates this issue.

Example 8. *Figure 13 shows the coefficient of variation for the 46 programs from the JOTAI collection that run the largest number of instructions when compiled with* `clang -O2`*. The mean running time of these programs (arithmetic averages of 10 samples) varies from 1.25 microseconds to 17 milliseconds. The*

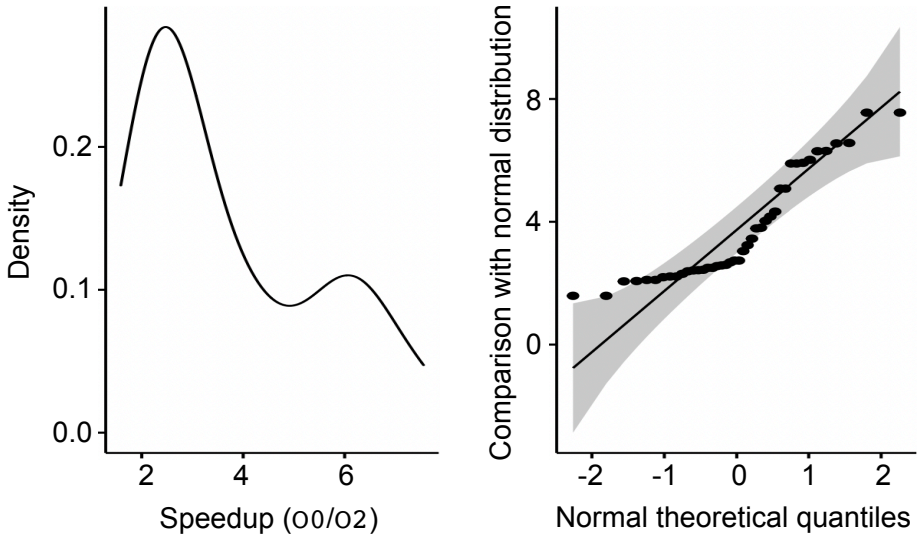

Figure 12: Density plot and QQ-plot for the 43 programs from SPEC17. Mean = 3.70x, Median = 2.72x, Outliers = 1.

coefficient of variation of the 5 fastest programs is always above 0.3 (i.e., 30%) and always below 0.03 (i.e., 3%) for the 5 slowest programs.

Despite these challenges, JOTAI can still be used in experiments that focus on running time. While the execution time of JOTAI benchmarks may fluctuate, the number of instructions they execute remains fixed, as the programs are deterministic and consist solely of visible instructions. This property motivates the following research question:

Question 3 (RQ3). *How strong is the correlation between the number of instructions executed by a JOTAI benchmark and its running time?*

Benchmarks: The 46 programs seen in Example 8[4].

Hardware: We measure correlations in two processors:

x86: i7-6700T, at 2.80GHz, with 7.6GB of RAM

[4]Initially, we tried to use 50 programs; however, 4 of them could not be used with VALGRIND in the Odroid board, due to excessive memory consumption. Without VALGRIND they work correctly.

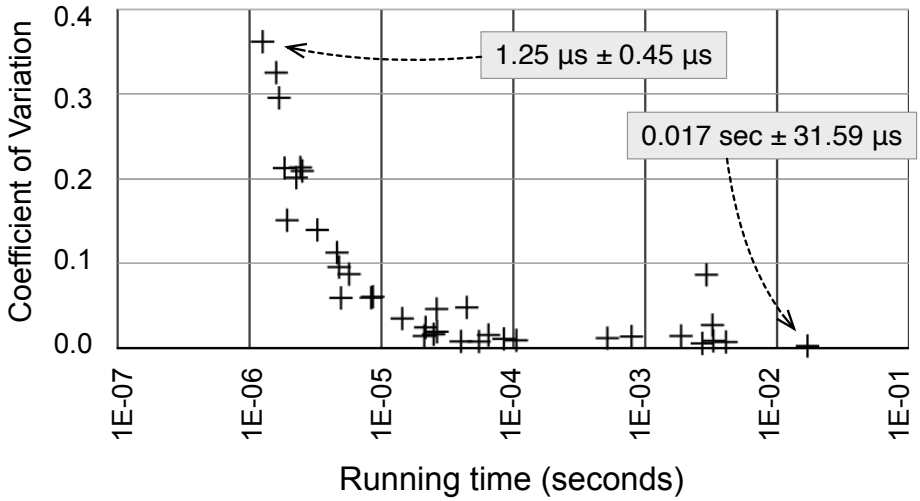

Figure 13: Coefficient of variation for the 50 programs that run more instructions when compiled with `clang -O0` in the Jotai collection.

arm: Odroid XU4 A15, at 2.00GHz, with 2GB of RAM

Both processors contain eight cores; however, programs run sequentially, at maximum frequency.

Methodology: We average the time of 10 executions of each benchmark, compiled with `clang -O0` or `clang -O2`. We count the number of instructions fetched per benchmark using CfgGrind. Running time and instructions refer to the function that constitutes the benchmark—the rest of the driver is not analyzed.

Discussion: Figure 14 plots the running time of programs versus the number of instructions they fetch. In contrast to running time, the number of instructions executed is fixed per benchmark. Figure 14 uses log scale in both axes; hence, it gives the false impression that programs compiled at `-O0` fetch as many instructions as programs compiled with `clang -O2`. However, as already seen in Section 4.2, this difference is large. To emphasize this distance, Figure 15 summarizes all the populations displayed in Figure 14.

Figure 16 shows the Spearman and the Kendall Rank Correlation Coefficients between the running time and the number of instructions executed per bench-

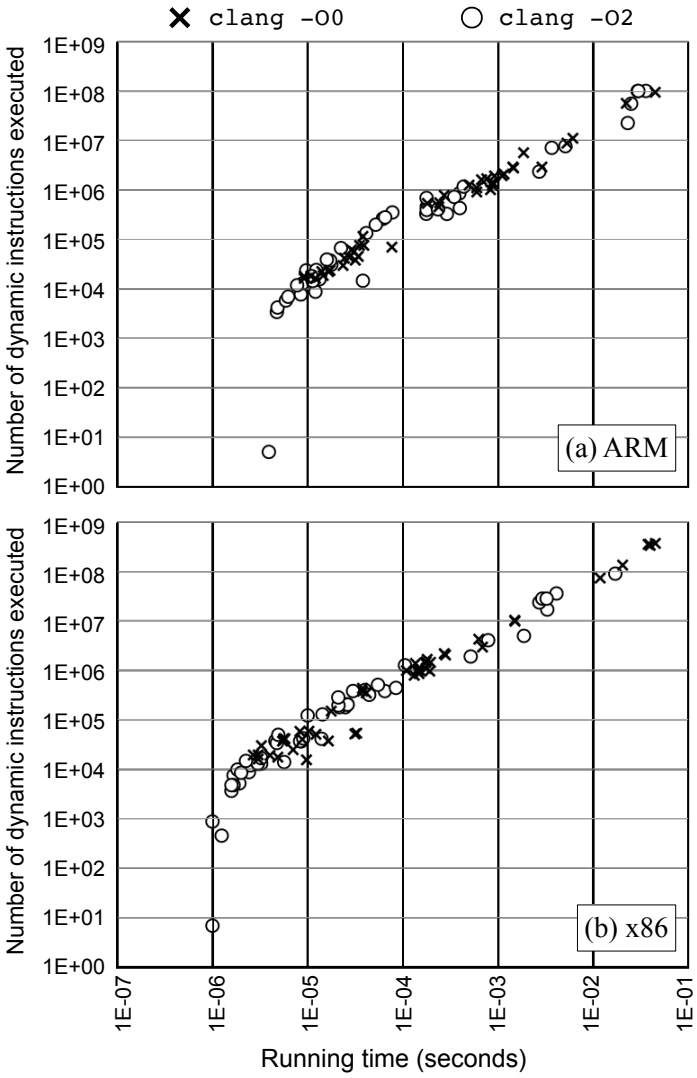

Figure 14: The correlation between the running time and the number of instructions executed per benchmark in two different architectures: (a) ARM and (b) x86, considering programs compiled at two different optimization levels.

mark. Both ranks are non-parametric; hence, recommended in the analysis of data that comes from non-normal distributions. In both cases, the correlation is very high, tending to 1.0. We have omitted Pearson's Coefficient, which is not robust in the face of outliers. Nevertheless, Pearson's Coefficient is also

	clang -O0		clang -O2		
	time (sec)	**# instrs.**	**time (sec)**	**# instrs.**	
Sum	1.0193	2.62E+09	0.1882	5.04E+08	**ARM A15** (2.00GHz)
Avg	0.0222	5.69E+07	0.0041	1.09E+07	
Min	9.17E-06	16,372	3.92E-06	5	
Max	0.2322	5.99E+08	0.0359	1.00E+08	
Sum	0.1997	1.65E+09	0.0372	2.45E+08	**x86 i7** (2.80GHz)
Avg	0.0048	3.93E+07	0.0009	5.83E+06	
Min	4.00E-06	15,304	1.00E-06	7	
Max	0.0448	3.66E+08	0.0171	9.26E+07	

Figure 15: Summary of data used to produce Figure 14.

	ARM A15 (2.00GHz)		**x86 i7** (2.80GHz)	
	`clang -O0`	`clang -O2`	`clang -O0`	`clang -O2`
Spearman	0.9642	0.9876	0.9903	0.9052
Kendall	0.8584	0.9206	0.9333	0.9791

Figure 16: Non-parametric correlation ranks between running time and number of instructions executed for the benchmarks in Figure 14.

greater than 0.9 in all four cases. Notice that although absolute times tend to be very different in the two boards, speedup ratios of `clang -O2` over `clang -O0`, considering averages, sums, maximums, and minimums tend to be very similar.

4.4. Coverage

Most of the programs that JOTAI produces are very simple: their execution amounts to a linear path of basic blocks. Nevertheless, some large programs can be found in this collection. This section provides the reader with some idea about the structural properties of these programs, namely, the average number of basic blocks visited and the proportion of branches traversed during the execution of benchmarks.

Question 4 (RQ4). *What is the expected size of* JOTAI *benchmarks, and what is the portion of this size that can be covered by simple constraints?*

Benchmarks: The 856 programs in the **DynGr** collection described in Section 4.2.

Hardware: Intel i7-6700T, at 2.80GHz, with 7.6GB of RAM

Software: Same apparatus seen in Section 4.2.

Methodology: We use CFGGRIND to count the number of basic blocks visited during the execution of each benchmark. CFGGRIND reconstructs the *dynamic slice* of the program. In other words, it builds the control-flow graph formed by the instructions fetched during the execution of the program. Such a dynamic slice is formed by *basic blocks*, i.e., maximal sequences of instructions that can execute in sequence, and *phantom blocks*, i.e., targets of branches that have not been visited. If a dynamic slice does not contain phantom blocks, then it has been completely visited during the execution of the program.

Discussion: Figure 17 shows the number of basic blocks visited and the number of phantom blocks observed during the execution of the 856 benchmarks in **DynGr** using the **big-arr** constraints. We show data for benchmarks compiled with different optimization levels of `clang`. On average, benchmarks have four to six basic blocks, with a median of five. The median value of phantom blocks is zero for most optimization levels. The number of benchmarks that are fully covered (i.e., that do not contain phantom blocks) varies per optimization level, peaking at 564 with `clang -Oz`.

Figure 18 summarizes the data presented in Figure 17. The row **sum** is the total of basic blocks visited during the execution of the benchmarks at different optimization levels or the total of different phantom blocks encountered during execution. Optimizations tend to reduce the number of basic blocks; however, this behavior is not always true: `clang -O3` increases the number of basic blocks, due to control-flow replication. Code vectorization, for instance, might replicate the body of loops. Nevertheless, although the static size of the program grows, its dynamic size—the number of instructions fetched—tends to decrease, as seen

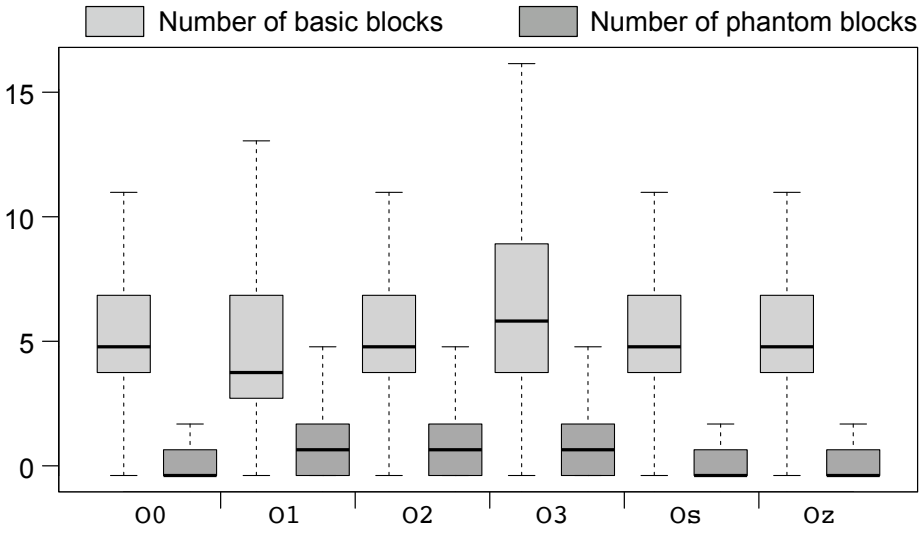

Figure 17: Number of basic blocks and phantom blocks found during the execution of the programs in **DynGr**.

in Section 4.2.

Figure 18 shows that the JOTAI methodology tends to generate non-trivial constraints, which guide execution through most paths within a program. However, the JOTAI collection also includes programs complex enough to contain multiple control-flow paths, some of which are not exercised by at least one input. This characteristic is particularly relevant, as it makes JOTAI programs valuable test cases for evaluating compiler-guided optimizations. This potential has recently been explored by Frenot and Pereira [34].

4.5. Case Study: Predicting the Speedup of Compiler Optimizations

One of the goals of JOTAI is to enable the construction of better models for predictive compilation. This section demonstrates this potential with a simple study to predict the *speedup* of compiler optimizations. Modern compilers, such as `clang`, provide different levels of optimization aimed at increasing the efficiency of executing code. The application of these optimizations can lead to a significant performance increase, commonly referred to as "speedup". However, the degree of this improvement can vary widely among different programs. In

	O0	O1	O2	O3	Oz	
median	5	4	5	5	5	
mean	5.999	5.143	5.359	6.764	5.395	**basic**
max	24	20	17	117	19	
sum	5,135	4,402	4,587	5,790	4,618	
median	0	1	1	1	0	
mean	0.7465	1.027	0.9206	1.68	0.4579	**phantom**
max	15	6	6	20	5	
sum	639	879	788	1,438	392	
full	489	424	415	283	564	

Figure 18: Summary of the data used to produce Figure 17 (The 856 programs in the **DynGr** dataset introduced in Section 4.2). **Full** is the number of programs that did not contain phantom nodes, i.e., fetched branches with untaken paths.

this section, we analyze the relative accuracy of three very simple models to predict the speedup of standard compiler optimizations. This kind of prediction requires a large number of executable programs—the goal of a collection such as JOTAI.

Question 5 (RQ5). *Can the analysis of source code attributes represented as histograms of opcodes lead to accurate predictions of the speedup from compiler optimizations?*

Benchmarks: The collection of 31,328 programs produced with different constraints, as seen in Figure 9.

Hardware: Intel i7-6700T, at 2.80GHz, with 7.6GB of RAM

Software: We measure the performance of programs as the number of instructions that these programs execute. As seen in Section 4.3, the number of instructions fetched is a good surrogate for running time—with the added benefit of being invariant across multiple executions of the same program. Instructions are counted using CFGGRIND [32], a VALGRIND [52] plugin, again,

following the methodology discussed in Section 4.3. The predictive models use the implementation of linear regression provided by Python's KERAS.

Methodology: This section evaluates three simple prediction models. We shall use the following methodology: given the universe of 31,328 available programs, 75% of them are separated as a *Training Set*. The rest is used as a `Test Set`. The model predicts speedups of optimizations by analyzing the training set and applies these predictions on the programs in the test set. The "speedup" is defined as the ratio between the number of instructions executed by a program when compiled with `clang -O0` and when compiled with `clang -O1`. The predictive models analyzed are described below:

Geo-Mean: the geometric mean of all the speedups observed in the training set are used as the prediction of speedups in the test set.

Model_O0: the histogram of source-code features is used as the input for linear regression. These histograms are extracted from the LLVM intermediate representation of programs compiled with `clang -O0`.

Model_O0_O1: two histograms of source-code features are used as the input for linear regression. These histograms are extracted from the LLVM intermediate representation of programs compiled with `clang -O0` and with `clang -O1`. Thus, these histograms have twice the length of those used in `Model_O0`.

The histograms used in `Model_O0` contain 69 entries: 64 entries to represent the different LLVM opcodes, four entries to represent loop depths (number of loops with depth one, depth two, depth three and depth four or more) and number of basic blocks. The histograms used in `Model_O0_O1` contain the same entries, although each appears twice: once as produced via `clang -O0`, and once as produced via `clang -O1`. To assess the relative performance of the predictive models, the geometric mean of the speedups obtained from the dataset served as a baseline for comparison. The models were evaluated using Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics quantify the

discrepancy between predicted and actual values, with lower values indicating superior precision.

Discussion: Figure 19 summarizes the results observed in this experiment. The findings indicate that linear regression built on histograms of opcodes are more precise to predict the speedup of compiler optimizations than the geometric mean of these optimizations observed on a training set. The histogram-based models outperform the geometric model in terms of both error metrics, irrespective of the dataset used for training. Furthermore, the comparison between `Model_O0_O1` and `Model_O0` suggests that including histograms of programs compiled with `clang -O1` improves the model's prediction capacity. In other words, the accuracy of a predictor that does linear regression on histograms of opcodes is more than twice higher than the accuracy of a predictor that uses a simple constant: the geometric mean of speedups observed on a training set.

	Mean Squared Error	Mean Absolute Error
`Model_O0`	0.00701	0.05353
`Model_O0_O1`	0.00592	0.04605
`Geometric mean`	0.01561	0.10137

Figure 19: Precision of different models used to predict the speedup of programs compiled with `clang -O1` over the same programs compiled with `clang -O0`.

These results suggest that histograms derived from opcode identifiers, loop depths, and the number of basic blocks in the source code can be used to predict the speedup resulting from the application of standard compiler optimizations. Histograms are simple data structures: they can be extracted from programs via a linear pass over the program's code. Notice that the observation that histograms of opcodes bring useful information to predict the behavior of programs is not new. Recently, Damásio et al. [55] and Gorchakov et al. [56] have shown that histograms can be effectively used to determine the algorithm that a program implements, given a number of candidates.

5. Related Work

The development of compilers requires benchmarks. Thus, some of the most celebrated papers in programming languages describe benchmark suites, such as Spec CPU2006 [57], MiBench [26], Rodinia [58], etc. These benchmarks are manually curated and typically comprise a small number of programs. Recently, Cummins et al. [20] have demonstrated that this reduced size fails to cover the space of program features that a compiler is likely to explore during its lifetime. Thus, researchers and enthusiasts have been working to generate a large number of diverse and expressive benchmarks. This section covers some of these efforts.

Random Synthesis. The generation of benchmarks for tuning predictive compilers has been an active field of research in the last ten years. Initial efforts directed at the development of predictive optimizers would use synthetic benchmarks conceived to find bugs in compilers. Examples of such synthesizers include CSmith [41], LDRGen [59] and Orange3 [60, 61]. Although conceived as test-case generators, these tools have also been used to improve the quality of the optimized code emitted by mainstream C compilers [62, 63]. Even CompilerGym provides randomly-produced CSmith programs. However, more recent developments indicate that synthetic codes tend to reflect poorly the behavior of human-written programs; hence, yielding deficient training sets [10, 64].

Guided Synthesis. Several research groups have used guided approaches to synthesize benchmarks [20, 27, 65, 66]. These techniques might rely on a template of acceptable codes, like Deniz et al. [66] do, or might use a machine-learning model to steer the generation of programs, like Cummins et al. [20] or Berezov et al. [27] do. Synthesis is restricted to a particular domain, like OpenCl kernels [20, 66]; or regular loops [27]. The approach described in this paper is different in the sense that the programs in Jotai are not synthesized; rather, they are mined from open-source repositories.

Code Mining. This paper produces benchmarks out of code from open-source repositories. We follow the methodology introduced by Faustino et al. [10] to ex-

tract and reconstruct programs, as Figure 1 illustrates. There exists a large body of literature about scraping programs from repositories. Some of these works aim at generating benchmarks to feed machine-learning models [67, 68, 69]; however, to the best of our knowledge, only Faustino et al. [10] and Armengol et al. [29] mine compilable programs to autotune compilers. In contrast, benchmarks used to train large-language models for code generation are formed by program snippets that are meant to be parsable, but not necessarily compilable [70, 69]. Notice that open-source repositories are not the only source of benchmarks to populate such models. For instance, Richards *et al.* have produced realistic JavaScript benchmarks, out of monitored browser sections [71]. A shortcoming of Richards's [71] approach is scalability: a human being is still required to create a browsing section that will give origin to one benchmark.

Generation of Executable Benchmarks. Many artificial benchmarks execute [29, 27, 20, 15, 16]. However, except for Berezov et al. [27]—which generates specific-domain loops— these collections follow CLDrive's [20] approach to filter out incorrect kernels. In the words of Tsimpourlas et al. [15]: "*[CLDrive] rejects kernels that (i) produce runtime errors [observable crashes]; (ii) do not modify any of the inputs (no output) or (iii) modify them differently for each run (not deterministic)*". Notice that this approach still leaves room for undefined behavior. Indeed, all our attempts to run benchmarks produced by Berezov [27] and Armengol [29] stumbled on undefined behaviors, which were reported (and confirmed) by these researchers. These previous generators also do not provide users with a way to explore the space of valid inputs, like the DSL that we introduce in Section 3.2—rather, the input generator is hardcoded into the synthesizer. As an example, Cummins et al.'s [20] CLDrive uses only one approach to produce inputs, which is similar to the **big-arr** constraint described in Section 4.2.

6. Conclusion

This paper introduced JOTAI: a set of principles, techniques and tools to generate executable C benchmarks that run without undefined behavior. JOTAI programs are mined from open-source repositories, compiled using inferred types, and equipped with inputs synthesized under user-defined constraints. Undefined behavior is filtered using both ADDRESSSANITIZER and KCC, which catch different classes of errors.

Our evaluation demonstrates that JOTAI can reliably produce sound executable benchmarks at scale. From over 70,000 candidate functions, we produced more than 31,000 benchmarks that pass both sanitization tools. To illustrate practical applications of these benchmarks, this paper has used them in a number of ways, which include revealing that optimization speedups do not follow a normal distribution, that instruction counts correlate strongly with runtime across architectures, and that even simple constraints yield high control-flow coverage. These findings validate JOTAI's utility for tasks such as statistical performance analysis and predictive compilation.

Programs in JOTAI can be used in a variety of ways: from stress testing processors and compilers to autotuning compilation tasks, as seen in Section 4.5. Currently, JOTAI programs are distributed as part of a standalone repository[5], or as a COMPILERGYM dataset. Our research group has been using JOTAI in various ways. For instance, JOTAI provided the benchmarks to evaluate MERLIN, a tool that infers the asymptotic complexity of programs [37], and to evaluate NISSE, an instrumentation-based profiler [43]. We are also aware of use cases beyond our research group. As an example, Meusel [72] has used JOTAI to tune his machine learning model to emulate the results of a probabilistic points-to analysis. Similarly, Krister Walfridsson has used JOTAI programs to test PYSMTGCC, a translation-validator for GCC[6]. In his words, "*JOTAI detected*

[5] JOTAI programs are available at `https://github.com/lac-dcc/jotai-benchmarks`

[6] For a brief description of PYSMTGCC, visit `https://kristerw.github.io/2022/09/13/translation-validation/`

cases that were missing in PySmtGcc [73]."

Acknowledgments

The authors would like to thank CNPq (grants 314645/2020-9 and 406377/2018-9), FAPEMIG (grant PPM-00333-18) and CAPES (Edital PrInt) for making this project possible.

Declaration of generative AI and AI-assisted technologies in the writing process. During the preparation of this work the authors used ChatGPT 3.5 in order to revise the grammar and the style of parts of the text. After using this tool, the authors reviewed and edited the content as needed. They take full responsibility for the content of the publication.

References

[1] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, D. Scheeff, Learning to schedule straight-line code, in: NIPS, MIT Press, Cambridge, MA, USA, 1997, p. 929–935. doi:10.5555/3008904.3009034.

[2] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, H. A. G. Wijshoff, A feasibility study in iterative compilation, in: Proceedings of the Second International Symposium on High Performance Computing, ISHPC '99, Springer-Verlag, Berlin, Heidelberg, 1999, p. 121–132.

[3] A. McGovern, E. Moss, Scheduling straight-line code using reinforcement learning and rollouts, in: NIPS, MIT Press, Cambridge, MA, USA, 1999, p. 903–909. doi:10.5555/340534.340836.

[4] M. E. Wolf, D. E. Maydan, D.-K. Chen, Combining loop transformations considering caches and scheduling, Int. J. Parallel Program. 26 (4) (1998) 479–503.

[5] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, Predictive modeling methodology for compiler phase-ordering, in: PARMA-DITAM, Association for Computing Machinery, New York, NY, USA, 2016, p. 7–12. doi:10.1145/2872421.2872424.

[6] C. Blackmore, O. Ray, K. Eder, Automatically tuning the GCC compiler to optimize the performance of applications running on the ARM cortex-m3, CoRR abs/1703.08228 (2017). arXiv:1703.08228.

[7] A. Brauckmann, A. Goens, S. Ertel, J. Castrillon, Compiler-based graph representations for deep learning models of code, in: CC, Association for Computing Machinery, New York, NY, USA, 2020, p. 201–211. doi:10.1145/3377555.3377894.

[8] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, H. Leather, ProGraML: A graph-based program representation for data flow analysis and compiler optimizations, in: ICML, Vol. 139, PMLR, Baltimore, Maryland, USA, 2021, pp. 2244–2253.

[9] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, H. Leather, CompilerGym: Robust, performant compiler optimization environments for AI research, CoRR abs/2109.08267 (2021). arXiv:2109.08267.

[10] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimarães, F. M. Q. Pereira, AnghaBench: A suite with one million compilable C benchmarks for code-size reduction, in: CGO, IEEE, Los Alamitos, CA, USA, 2021, pp. 378–390. doi:10.1109/CGO51591.2021.9370322.

[11] A. Faustino, E. Borin, F. M. Quintão Pereira, O. Nápoli, V. Rosário, New optimization sequences for code-size reduction for the LLVM compilation infrastructure, in: SBLP, ACM, New York, NY, USA, 2021, p. 33–40. doi:10.1145/3475061.3475085.

[12] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, in: AAAI, AAAI Press, Phoenix, Arizona, 2016, p. 1287–1293. doi:10.5555/3015812.3016002.

[13] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, F. Reiss, Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks (2021). arXiv:2105.12655.

[14] S. H. H. Ding, B. C. M. Fung, P. Charland, Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: SP, IEEE, Washington, DC, USA, 2019, pp. 472–489. doi:10.1109/SP.2019.00003.

[15] F. Tsimpourlas, P. Petoumenos, M. Xu, C. Cummins, K. M. Hazelwood, A. Rajan, H. Leather, Benchpress: A deep active benchmark generator, in: A. Klöckner, J. Moreira (Eds.), Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022, ACM, 2022, pp. 505–516. doi:10.1145/3559009.3569644.
URL https://doi.org/10.1145/3559009.3569644

[16] F. Tsimpourlas, P. Petoumenos, M. Xu, C. Cummins, K. Hazelwood, A. Rajan, H. Leather, Benchdirect: A directed language model for compiler benchmarks (2023). arXiv:2303.01557.

[17] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: ICSE, IEEE Press, New York, NY, US, 2019, p. 783–794. doi:10.1109/ICSE.2019.00086.

[18] G. Zhao, J. Huang, DeepSim: Deep learning code functional similarity, in:

ESEC/FSE, ACM, New York, NY, USA, 2018, p. 141–151. doi:10.1145/3236024.3236068.

[19] Z. Wang, M. F. P. O'Boyle, Machine learning in compiler optimization, Proceedings of the IEEE 106 (11) (2018) 1879–1901. doi:10.1109/JPROC.2018.2817118.

[20] C. Cummins, P. Petoumenos, Z. Wang, H. Leather, Synthesizing benchmarks for predictive modeling, in: CGO, IEEE, Piscataway, NJ, USA, 2017, pp. 86–99. doi:10.1109/CGO.2017.7863731.

[21] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, F. Reiss, Project CodeNet (2021). doi:10.5281/zenodo.4814769.

[22] A. Grossman, L. Paehler, K. Parasyris, T. Ben-Nun, J. Hegna, W. Moses, J. M. M. Diaz, M. Trofin, J. Doerfert, Compile: A large ir dataset from production sources (2023). arXiv:2309.15432.

[23] C. Hathhorn, C. Ellison, G. Roşu, Defining the undefinedness of c, in: PLDI, Association for Computing Machinery, New York, NY, USA, 2015, p. 336–345. doi:10.1145/2737924.2737979.

[24] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, M. F. Kaashoek, Undefined behavior: what happened to my code?, in: Asia-Pacific Workshop on Systems, 2012, pp. 1–7.

[25] G. Fursin, Collective tuning initiative, CoRR abs/1407.3487 (2014). arXiv:1407.3487.
URL http://arxiv.org/abs/1407.3487

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: WWC, IEEE, Washington, DC, USA, 2001, pp. 3–14. doi:10.1109/WWC.2001.15.

[27] M. Berezov, C. Ancourt, J. Zawalska, M. Savchenko, COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks, in: F. Palumbo, J. a. Bispo, S. Cherubin (Eds.), PARMA-DITAM, Vol. 100 of Open Access Series in Informatics (OASIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 3:1–3:14. doi:10.4230/OASIcs.PARMA-DITAM.2022.3.
URL https://drops.dagstuhl.de/opus/volltexte/2022/16119

[28] L. Bjertnes, J. O. Tørring, A. C. Elster, LS-CAT: A large-scale CUDA autotuning dataset, CoRR abs/2103.14409 (2021). arXiv:2103.14409.
URL https://arxiv.org/abs/2103.14409

[29] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, M. F. P. O'Boyle, ExeBench: An ML-scale dataset of executable C functions, in: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, MAPS 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 50–59. doi:10.1145/3520312.3534867.
URL https://doi.org/10.1145/3520312.3534867

[30] L. T. C. Melo, R. G. Ribeiro, M. R. de Araújo, F. M. Q. Pereira, Inference of static semantics for incomplete C programs, Proc. ACM Program. Lang. 2 (POPL) (2018) 29:1–29:28. doi:10.1145/3158117.

[31] L. T. C. Melo, R. G. Ribeiro, B. C. F. Guimarães, F. M. Q. Pereira, Type inference for C: Applications to the static analysis of incomplete programs, ACM Trans. Program. Lang. Syst. 42 (3) (2020). doi:10.1145/3421472.

[32] A. Rimsa, J. N. Amaral, F. M. Q. Pereira, Practical dynamic reconstruction of control flow graphs, Softw. Pract. Exp. 51 (2) (2021) 353–384. doi:10.1002/spe.2907.
URL https://doi.org/10.1002/spe.2907

[33] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, Addresssanitizer:

A fast address sanity checker, in: ATC, USENIX Association, USA, 2012, p. 28.

[34] L. Frenot, F. M. Q. a. Pereira, Reducing the overhead of exact profiling by reusing affine variables, in: Compiler Construction, CC 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 150–161. doi:10.1145/3640537.3641569.
URL https://doi.org/10.1145/3640537.3641569

[35] M. Canesche, et al., Optimizing machine learning models: a droplet search approach to efficient kernel scheduling, Ph.D. thesis, Universidade Federal de Minas Gerais (2024).

[36] R. P. Munafo, Machine learning enhanced code optimization for high-level synthesis (ml-ecohs), Ph.D. thesis, Boston University (2024).

[37] R. Sumitani, L. Silva, F. Campos, F. Pereira, A class of programs that admit exact complexity analysis via newton's polynomial interpolation, in: SBLP, Association for Computing Machinery, New York, NY, USA, 2023, p. 50–55. doi:10.1145/3624309.3624311.
URL https://doi.org/10.1145/3624309.3624311

[38] E. Vatai, A. Drozd, I. R. Ivanov, J. E. Batista, Y. Ren, M. Wahib, Tadashi: Enabling ai-based automated code generation with guaranteed correctness, arXiv preprint arXiv:2410.03210 (2024).

[39] I. R. Ivanov, J. Meyer, A. Grossman, W. S. Moses, J. Doerfert, Input-gen: Guided generation of stateful inputs for testing, tuning, and training, arXiv preprint arXiv:2406.08843 (2024).

[40] A. R. Álvares, J. N. Amaral, F. M. Q. Pereira, Instruction visibility in SPEC CPU2017, J. Comput. Lang. 66 (2021) 101062. doi:10.1016/j.cola.2021.101062.
URL https://doi.org/10.1016/j.cola.2021.101062

[41] X. Yang, Y. Chen, E. Eide, J. Regehr, Finding and understanding bugs in c compilers, in: PLDI, ACM, New York, NY, USA, 2011, pp. 283–294. doi:10.1145/1993498.1993532.

[42] C. Kind, J. Coelho, B. Kind, F. Pereira, Geração automática de benchmarks para compilação preditiva, in: SBLP, Association for Computing Machinery, New York, NY, USA, 2022, p. 59–67. doi:10.1145/3561320.3561323.
URL https://doi.org/10.1145/3561320.3561323

[43] L. Frenot, F. M. Q. Pereira, Reducing the overhead of exact profiling by reusing affine variables, accepted in the International Symposium on Compiler Construction (2023).

[44] T. Freeman, F. Pfenning, Refinement types for ml, in: PLDI, Association for Computing Machinery, New York, NY, USA, 1991, p. 268–277. doi:10.1145/113445.113468.
URL https://doi.org/10.1145/113445.113468

[45] C. da Linguagem, Defect report #260, jTC 1, SC 22, WG 14 (2004).
URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm

[46] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, P. Sewell, Into the depths of c: Elaborating the de facto standards, in: PLDI, Association for Computing Machinery, New York, NY, USA, 2016, p. 1–15. doi:10.1145/2908080.2908081.
URL https://doi.org/10.1145/2908080.2908081

[47] B. Zorn, Comparing mark-and sweep and stop-and-copy garbage collection, in: LFP, Association for Computing Machinery, New York, NY, USA, 1990, p. 87–98. doi:10.1145/91556.91597.
URL https://doi.org/10.1145/91556.91597

[48] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, P. Bertelsen, Programming with regions in the ML Kit, DIKU Rapport 97 (1997) 12.

[49] D. R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, Software: Practice and Experience 20 (1) (1990) 5–12.

[50] S. J. Cyvin, Algorithm 226: Normal distribution function, Commun. ACM 7 (5) (1964) 295. doi:10.1145/364099.364315.
URL https://doi.org/10.1145/364099.364315

[51] J. Bucek, K.-D. Lange, J. v. Kistowski, Spec cpu2017: Next-generation compute benchmark, in: ICPE, Association for Computing Machinery, New York, NY, USA, 2018, p. 41–42. doi:10.1145/3185768.3185771.
URL https://doi.org/10.1145/3185768.3185771

[52] N. Nethercote, J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, in: PLDI, Association for Computing Machinery, New York, NY, USA, 2007, p. 89–100. doi:10.1145/1250734.1250746.
URL https://doi.org/10.1145/1250734.1250746

[53] B. Wescott, Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With, 1st Edition, CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2013.

[54] A. F. da Silva, B. Kind, J. W. M. aes, J. Rocha, B. G. aes, F. M. Q. Pereira, AnghaBench: a synthetic collection of benchmarks mined from open-source repositories, Tech. Rep. 01-2020, Universidade Federal de Minas Gerais (2020).

[55] T. Damásio, M. Canesche, V. Pacheco, M. Botacin, A. Faustino da Silva, F. M. Quintão Pereira, A game-based framework to compare program classifiers and evaders, in: Proceedings of the 21st ACM/IEEE International

Symposium on Code Generation and Optimization, CGO 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 108–121. doi:10.1145/3579990.3580012.
URL https://doi.org/10.1145/3579990.3580012

[56] A. V. Gorchakov, L. A. Demidova, P. N. Sovietov, Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task, Future Internet 15 (9) (2023). doi:10.3390/fi15090314.
URL https://www.mdpi.com/1999-5903/15/9/314

[57] J. L. Henning, SPEC CPU2006 benchmark descriptions, SIGARCH Comput. Archit. News 34 (4) (2006) 1–17. doi:10.1145/1186736.1186737.

[58] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IISWC, IEEE, Washington, DC, USA, 2009, pp. 44–54. doi:10.1109/IISWC.2009.5306797.

[59] G. Barany, Liveness-driven random program generation, in: LOPSTR, Springer, Heidelberg, Germany, 2017, pp. 112–127. doi:10.1007/978-3-319-94460-9_7.

[60] E. Nagai, A. Hashimoto, N. Ishiura, Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions, IPSJ Trans. System LSI Design Methodology 7 (2014) 91–100.

[61] K. Nakamura, N. Ishiura, Introducing loop statements in random testing of c compilers based on expected value calculation (2015).

[62] A. Hashimoto, N. Ishiura, Detecting arithmetic optimization opportunities for c compilers by randomly generated equivalent programs, IPSJ Transactions on System LSI Design Methodology 9 (2016) 21–29.

[63] G. Barany, Finding missed compiler optimizations by differential testing, in: Proceedings of the 27th International Conference on Compiler

Construction, CC 2018, ACM, New York, NY, USA, 2018, pp. 82–92. doi:10.1145/3178372.3179521.

[64] A. Goens, A. Brauckmann, S. Ertel, C. Cummins, H. Leather, J. Castrillon, A case study on machine learning for synthesizing benchmarks, in: MAPL, ACM, New York, NY, USA, 2019, pp. 38–46. doi:10.1145/3315508.3329976.

[65] A. Chiu, J. Garvey, T. S. Abdelrahman, Genesis: A language for generating synthetic training programs for machine learning, in: CF, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–8. doi:10.1145/2742854.2742883.
URL https://doi.org/10.1145/2742854.2742883

[66] E. Deniz, A. Sen, Minime-gpu: Multicore benchmark synthesizer for gpus, ACM Trans. Archit. Code Optim. 12 (4) (nov 2015). doi:10.1145/2818693.
URL https://doi.org/10.1145/2818693

[67] J. Fowkes, C. Sutton, Parameter-free probabilistic api mining across github, in: FSE, Association for Computing Machinery, New York, NY, USA, 2016, p. 254–265. doi:10.1145/2950290.2950319.
URL https://doi.org/10.1145/2950290.2950319

[68] G. Gousios, D. Spinellis, Mining software engineering data from github, in: ICSE-C, IEEE Press, Washington, DC, US, 2017, p. 501–502. doi:10.1109/ICSE-C.2017.164.
URL https://doi.org/10.1109/ICSE-C.2017.164

[69] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, D. Poshyvanyk, Evaluating and explaining large language models for code using syntactic structures (2023). arXiv:2308.03873.

[70] P. Vaithilingam, T. Zhang, E. L. Glassman, Expectation vs. experience: Evaluating the usability of code generation tools powered by large language

models, in: CHI EA, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–7. doi:10.1145/3491101.3519665.
URL https://doi.org/10.1145/3491101.3519665

[71] G. Richards, A. Gal, B. Eich, J. Vitek, Automated construction of javascript benchmarks, SIGPLAN Not. 46 (10) (2011) 677–694.

[72] S. Meusel, Learning a graph neural network based model for probabilistic alias analysis, bachelor Thesis (2022).

[73] K. Walfridsson, Personal communication, sent on October 29th (2022).